9.1 An Introduction to Abstraction

Abstraction is fundamental to our everyday lives, not just in computing. Loosely, abstraction is about understanding how to use something without knowing how it works. Consider your refrigerator—how does it work? Does it matter? We know that we can open a fridge door, place something (probably food) inside, and the fridge will keep it cold. So our notion of a fridge is really quite abstract; there are many thousands of refrigerator types, each one designed and built by different companies and people around the world. But this is irrelevant: when you go to a friend's house, you can use their fridge just as you would your own, without any extra help.

There are several examples of abstraction in the real world. It doesn't matter how a watch was built, so long as we can use it to tell time. It doesn't matter how a cup was made or what materials it was made out of, so long as we can use it to hold liquid. Divorcing the nitty gritty details of how something works with how to use it is abstraction. And it is abstraction that has allowed for ingenuity and creativity to advance technology (i.e., how something works) without having to re-educate the entire world on how to use a cup.[1]

[1] Of course, humans have also creatively improved how we use things, like attaching a handle to a cup meant to contain hot coffee.

We can think of abstraction as allowing for the separation of two groups of people with different goals: the *creators* of an entity, and the *users* (or *clients*) of that entity. Sometimes there's overlap between these two groups, but much of the time—especially as technology and systems have grown more complex—these two groups are fairly separate. Creators are responsible for designing, building, and implementing an entity, and users are responsible for, well, using it.

The **interface** of an entity is the boundary between creator and user: it is the set of rules (implicit or explicit) governing how users can interact with that entity. We call an interface the *public* side of an entity; it is the part of the creator's work that everyone can interact it. Creators are responsible for the design of the interface, while users are responsible for learning the interface in order to interact with the entity. For example, the interface of a cup is how you use it: where to put liquid and where to hold it when taking a drink.

Abstraction in computer science

Abstraction and interfaces are incredibly useful concepts in computer science because of the complexity of programming languages, algorithms, and computer hardware that come with modern technology. We've been using abstraction all the way through this course, playing the role of creator in some cases, and users in others:

- We are *users* of the Python programming language itself, which provides an interface that hides the details of our computer hardware, processor instructions, memory, storage, and graphics.[2] While we have learned some details about how

[2] This isn't unique to Python, of course: *every* programming language is an interface between the programmer and their computer.

  the Python interpreter works (like our discussion of its array-based list implementation in the previous chapter), we've barely scratched the surface of this large and complex software.

- We are *users* of built-in Python functions, data types, and modules. We don't know how the creators of the Python programming language have implemented these built-ins, but have learned how to *use* them to write useful programs.

- We are *creators* of new Python functions, data types and modules. Each time you have followed the function or data class design recipe, you have created an interface.

  For a *function*, its interface is its header and docstring: these specify how to call the function, the arguments it expects, and what the function does. The function body, is the implementation of the function, and are not part of its interface—someone who wants to use our function should not have to look a the function body to determine what it does.

  For a *data class*, its interface is the name of the data class and the names and types of its attributes, and the class docstring. In other words, *every* part of what we write to define a new data class is part of its interface! How data classes are actually implemented has been hidden from us in the `@dataclass` decorator, though we'll begin learning about how this implementation works in the next section.

  Finally, the interface of a Python *module* is simply the collection of interfaces of the functions and data types defined in that module, plus any additional documentation in the module docstring. For every Python file you've written so far, you've created a module that could be used by other programmers.

- When studying mathematical statements, we have acted as both *users* and *creators*. Every time we write a proof, we act as a creator of knowledge, providing airtight evidence that a statement is True.[3] Every time we use an "external statement" in a

[3] You can view a proof as an "implementation" of a statement. Rather than acting as a set of instructions to execute to complete a task, a proof is a set of statements that assert the truth of a statement.

  proof, like the Quotient-Remainder Theorem or Fermat's Little Theorem, we are acting as *users* of these statements, and do not worry about how they have been proved.

Interfaces are contracts

As we work with more and more programming interfaces—different functions, data types, modules, and even programming languages—we see just how challenging designing interfaces can be. Every interface is a contract between creator and user: while creators have control over how they design an interface, they have the responsibility to make that interface easy and intuitive for users. Good interfaces are simple and strive to minimize the cognitive load on users; bad interfaces are cumbersome, ambiguous, and require the user to keep track of many unrelated details. Because interfaces are public, as creators we put a lot of effort into designing good interfaces, a topic we'll discuss in this year but that you'll explore far more in future courses.

Moreover, because interfaces are contracts, they are hard to change once released—made public to users—as any change will have ramifications on every user. We have been the users used several Python modules so far, such as `timeit`, `pytest`, and `doctest`. What would happen if the creators of one of these modules decided to make a change to that interface, like changing the `timeit` function name to `time_it`? This one character change would cause all code that uses the `timeit.timeit` function to no longer work! As clients of the `timeit` module, we would not be very happy.

There are two sides to every contract. Just as creators are beholden to keep the interface they provide, users are limited to that interface as well. When we act as the creators of a function or module, we are free to modify their implementations in any way we wish, as long as we do not change the public interface. We can fix a bug, simplify the code, or use a more efficient algorithm, all to improve our implementation *without* affecting our users. In software engineering, it is important to clearly define what the public interface of a piece of code actually is, so that its creators know precisely what they must preserve and what they are free to change.

Over the next two chapters, we'll explore the concepts of abstraction, public interfaces, and private implementations in more detail. We'll study how we can build our own Python data types from scratch (without relying on `@dataclass`) to gain full control over defining a data type's public interface. We'll create implementations of abstract data types and models of real-world domains, using the ideas we've introduced here to define clear public interfaces for every part of what we do.